IST-510255
EmBounded
Automatic Prediction of Resource Bounds for Embedded Systems

Specific Targeted Research Project (STReP)
FET Open

D42 (WP6): Certification Using Dependent Types

Due date of deliverable: February 2009
Actual submission date: March 2009

Start date of project: 1st March 2005 Duration: 48 months

Lead contractor: Ludwig-Maximilians Universität, München Revision: 1.13

Purpose: Dependent types offer a mechanism that allows verifiable properties to be associated with program text in a way that is easily verifiable by a fully automatic type checker. By using such an approach, it is possible both to extend the range of programs that can be verified, and to allow the programmer to influence the verification process. The purpose of this deliverable is to develop and evaluate prototype representations of costed-by-construction programs as developed as part of WP3/4.

Results: We have explored three methods for representing costed-by-construction programs in a dependently typed language, Idris. These are: a shallow embedding, in which costed programs are implemented directly in Idris with size annotations; a deep embedding, in which we implement a language with size annotations and its interpreter; and finally a hybrid approach in which we implement a domain specific embedded language in Idris, exploiting host language features as far as possible.

Conclusion: Each approach we have investigated has its own strengths and weaknesses. The shallow embedding is easy to implement, but has limited expressivity; the deep embedding is hard to implement with current technology, but is a worthy (and ambitious) ultimate goal as it would lead to fully machine checkable proofs of cost correctness. In this initial investigation, we have found the domain specific language approach to be a promising middle ground, combining features of the host language with manipulation of cost constraints in the state where required.

Project co-funded by the European Commission within the 6^{th} Framework Programme (2002-06)			
Dissemination Level			
PU	Public		⁎
PP	Restricted to other programme participants	(including the Commission Services)	
RE	Restricted to a group specified by the consortium	(including the Commission Services)	
CO	Confidential only for members of the consortium	(including the Commission Services)	

Certification Using Dependent Types

Edwin Brady <eb@cs.st-andrews.ac.uk>,
School of Computing Science, Univ of St Andrews, St Andrews, Scotland

Hans-Wolfgang Loidl <hwloidl@tcs.ifi.lmu.de>,
Institut für Informatik, Theoretische Informatik
Ludwig-Maximilians Universität, München

Abstract

We describe a prototype approach to representing costed-by-construction programs in a dependent type theory, exploiting information provided by cost analyses. We describe our implementation of dependent types, and show how the language interacts with an underlying theorem prover. We discuss the differences between shallow embeddings (adding cost annotations to existing programs in the host language) and deep embeddings (fully implementing the typing rules and an interpreter in the host language). We illustrate, using relatively simple cost analyses, how a complete trace of the proof of a size constraint leads to a verifiable program in a dependent type theory, and begin to explore how such a method can be applied to the cost analyses we have previously developed.

The positioning of this deliverable is discussed in Section 6.1.

Major Revisions		
Revision	Date	Changes
1.13	14 Aug. 2009	table-of-contents (addressing Review Report Year 4)
1.11	30 March 2009	initial version

Contents

1 Introduction

In this report, we describe a prototype approach to representing costed-by-construction programs in a dependent type theory. We have implemented a prototype dependently typed functional language (IDRIS) intended for linking into Hume's expression layer, and used it to investigate methods for representing explicitly costed functional programs. This was a roadmapping activity, intended to determine the feasibility of representing resource safe programs in a dependently typed core language with fully explicit proof. As such, in this report, we focus on:

- The language IDRIS itself.
- The representation of proofs and the link between the language and the underlying proof system.
- Methods for representing cost, using shallow and deep embeddings of costed programs in IDRIS, and using a Domain Specific Embedded Language (DSEL) approach.

This work is closely related to the resource certificates we have described in Deliverable D21 [LB08], and the cost analyses presented in Deliverables D5 [JLH07b], D11 [JLH07a] and D14 [JLH07c]. These relationships are discussed in more detail in Section 6. At this stage we have chosen to explore what is possible with the dependently typed representation in general, rather than being constrained by the results of previous analyses.

1.1 Dependent Types for Cost Analysis

Using dependent types as the intermediate representation has several potential benefits:

- We can verify the correctness of constraints given by an external inference system. A program with embedded resource usage constraints is a complete, self-contained, checkable term; correctness of the constraints is verified by the typechecker, a small and well understood (and hence relatively straightforward to verify) program. This design keeps the trusted code base small. We do not have to provide soundness or completeness proofs for our high level type systems if we implement it entirely within a system already known to be sound and complete.
- In situations where an inference system is not powerful enough to derive size constraints, or where the user requires a weaker constraint on the size, we can allow the user to specify the size constraint by hand and still be able to check it. Dependent types allow us to overcome the limitations of a sized type based inference system — it is always possible for the user to provide hints.
- Where an automated proof construction system is not powerful enough to solve a constraint, we can expose proof obligations to the user, either for the user to solve, or to show that a constraint cannot be satisfied.
- We can express more complex properties than those available in weaker sized type systems; we are not restricted in the constraint language. Since we can write programs at the type level, we can extend the constraint language as we wish. In particular, this gives us more flexibility in expressing the cost of higher order functions. There need be no loss of size information — we can give each program as precise a size predicate as we need.

It is important to note that we do not use the dependent type system to help *infer* size information; this is left to external cost analyses (studied in WP 3 and 4), which must provide fully explicit results. Rather, we use dependent types to certify that the constraints we have are satisfiable.

In this report, we will describe our implementation of dependent types, and show how the language interacts with an underlying theorem prover. We illustrate (using relatively simple cost analyses) how a complete trace of the proof of a size constraint leads to a verifiable program in a dependent type theory, using both a deep embedding (a full representation of typing rules using an interpreter for the embedded language) and a shallow embedding (directly adding cost annotations using the dependent type theory). We begin to explore how such a method can be applied to the cost analyses we have developed in D5 [JLH07b], D11 [JLH07a], D14 [JLH07c] and resource certificates developed in D21 [LB08].

2 The Idris Programming Language

Idris is an experimental functional programming language with dependent types. It is evaluated eagerly, and is implemented on top of the Ivor theorem proving library [Bra07], which gives access to an interactive tactic based theorem prover.

Idris serves as a prototype of a computation language for a dependently typed implementation of Hume. We have chosen to implement our own language for this purpose, rather than an existing tool such as Agda or Coq, since it gives us complete freedom to experiment with some of the finer details of the type system (e.g. whether all programs must be total, or whether general recursion is allowed in certain circumstances) and to make our own design and implementation choices without any constraints.

In this section, we will introduce the Idris programming language, and describe in particular its links with the underlying theorem prover. This will allow us to cleanly separate computational content (i.e. a user's program) from logical content (i.e. certificates for verifying program cost).

Idris uses Haskell-style basic type declarations, e.g.:

```
data Nat =     O   | S Nat;
data List a =  Nil | Cons a (List a);
```

We can define dependently typed vectors using a guarded abstract data type (GADT) style syntax (as used in Haskell), with `#` indicating the type of types, as follows:

```
data Vect  : # -> Nat -> # where
     VNil  : Vect A O
   | VCons : A -> (Vect A k) -> (Vect A (S k));
```

We can similarly define a dependent type of *finite sets*, which we can use in our programs to encode bounded numbers:

```
data Fin : Nat -> # where
     fO  : Fin (S k)
   | fS  : (Fin k) -> (Fin (S k));
```

`VCons` and `fS` illustrate the use of *implicit arguments*. Although `k` is not declared, it is clearly a `Nat` in each case, and this type can be inferred by unification. We could also write this explicitly as:

```
VCons : {k:Nat} -> A -> (Vect A k) -> (Vect A (S k));
```

where the braces indicate that `k` is implicit, and will therefore not be given explicitly to applications of `VCons`. Note that names can also be given to explicit arguments in this way — this is useful either where a later argument or return type depends on it, or where naming an argument helps the reader understand its purpose.

Sometimes we need to give the type of implicit arguments, either where unification is not strong enough to infer types, or in order to clarify types for the reader. Consider the definition of `ElemIs`, a predicate stating the location of an item in a vector:

```
data ElemIs : (Fin n) -> A -> (Vect A n) -> # where
   first : {xs:Vect A n} ->
           (ElemIs fO x (VCons x xs))
 | later : {xs:Vect A n} -> {i:Fin n} ->
           (ElemIs i x xs) ->
           (ElemIs (fS i) x (VCons y xs));
```

Here, we need to give types for `xs` and `i`, since they have types that depend on another variable `n` (which we do not declare here). These declarations are rather distracting, however, and we can therefore use global declarations like this:

```
using (xs:Vect A n, i:Fin n) {
  data ElemIs : (Fin n) -> A -> (Vect A n) -> # where
     first : (ElemIs fO x (VCons x xs))
   | later : (ElemIs i x xs) ->
             (ElemIs (fS i) x (VCons y xs));
}
```

The `using`-notation allows default types to be declared for implicit arguments, so de-cluttering the notation, and allowing only explicit arguments to be written. This is similar to a parametrised module in Coq or Agda, in that it associates a series of definitions with indices, but differs in that the Idris notation is purely syntactic.

2.1 Monads and `do`-notation

Idris does not (yet) support type classes or any equivalent type abstraction mechanism. While, in general, this does not cause serious difficulties, generic `do`-notation for monadic code is a valuable notational convenience and, in Haskell, this relies essentially on the use of type classes. At this stage of the Idris design, we have taken the simplest useful approach, which is to allow the programmer to explicitly define which monad is intended (with the `IO` monad the default), using a form of the `using`-notation we defined earlier. For example, `Maybe` is a monad:

```
data Maybe a = Nothing | Just a;

mbind : (Maybe a) -> (a -> (Maybe b)) -> (Maybe b);
mreturn :  a -> (Maybe a);
```

We can use `do`-notation for `Maybe` by declaring the operations to bind/return variables to be `mbind`/`mreturn`:

```
do using (mbind, mreturn) {
  addMaybe : (Maybe Int) -> (Maybe Int) -> (Maybe Int);
  addMaybe mx my = do { x <- mx; y <- my;
                        return (x+y); };
}
```

2.2 Theorem Proving

One of the benefits of being able to predicate types on values is that types can express theorems about programs. Idris has a built-in type for representing equality, with one constructor `refl`:

```
refl n : n=n;
```

While we can write down types such as `0=(S n)`, where the values are unequal, we can only construct an instance if the values really are equal, since the constructor `refl` has type `(n:A)->(n=n)`. Using this, we can write proofs of properties of programs. For example, consider the following definition of addition over natural numbers:

```
plus : Nat -> Nat -> Nat;
plus O y = y;
plus (S k) y = S (plus k y);
```

We can define and prove simple properties of this, such as that adding zero to a number has no effect:

```
eq_resp_S : (m=n) -> ((S m) = (S n));
eq_resp_S (refl n) = refl (S n);

plus_nO : (n:Nat) -> ((plus n O) = n);
plus_nO O = (refl O);
plus_nO (S n) = eq_resp_S (plus_nO n);
```

Using these simple properties, we can go on to define more complex properties, such as the commutativity or associativity of addition. We can also implement relations other than addition. In the context of resource limited systems, a property we may be interested in is that a number is smaller than another:

```
data LE : Nat -> Nat -> # where
   leO : LE O n
 | leS : (LE n m) -> (LE (S n) (S m));
```

An instance of `LE x y` can only be constructed if `x` really is less than or equal to `y`.

2.3 Interactive development

If we are to make full use of dependent types, it is necessary to support some form of interactive program development and theorem proving. While it is theoretically possible to write a full, type-correct program and associated proofs, it greatly aids both the construction of programs, and readability, if purely logical parts can be kept separate from computation. Being built on top of a theorem prover, Idris allows incomplete programs containing *metavariables*, standing for parts of programs to be filled in later. Before such programs can be run, the metavariables must be instantiated. Idris allows metavariables to be introduced in two ways:

1. Programs may contain **holes**, marked with a `?`. Holes stand for expressions whose type is known (inferred by the typechecker) but whose value is unknown.
2. Pattern-match clauses may be marked as **provisional**, by defining them using `?=` rather than `=`. Provisional clauses may be incorrectly typed, in which case the typechecker will insert a lemma (to be proved by the user) that will convert them to the correct type.

In this report, we will present programs using the first method only, but the libraries we have used in our research make use of both.

3 Cost Representation

In this section we present the various methods we have investigated for representing costs as dependent types, and discuss the relative merits of each. There are two distinct approaches we could take: a shallow embedding of terms in a dependently typed *host* language, or a deep embedding, using a dependently typed language to implement a costed language's abstract syntax and semantics.

3.1 Shallow Embeddings

With a shallow embedding, we use a dependently typed core language's type system and structure to capture the resource requirements, which allows programs to be represented and evaluated directly in the host language. In the context of Hume, successfully representing programs with costs as shallow embeddings would provide a strong case for using a dependently typed computation language. The theory behind the approach is outlined in [BH06]; in this section we describe the implementation in Idris.

The main idea of the embedding is that all values are indexed by a size metric, and values returned by a function additionally carry a proof of a size predicate, defined as follows:

```
data Size : (A:Nat->#) -> (P:(n:Nat)->(A n)->#) -> # where
  size : {A:Nat->#} -> {P:(n:Nat)->(A n)->#} ->
         (val:A n) -> (P n val) -> (Size A P);
```

For example, we may have a `List` datatype defined as follows:

```
data List a = Nil | Cons a (List a);
```

This value should be indexed by a size metric. For lists, the simplest size metric is their length (we could also, in more concrete terms, consider the amount of heap space a list occupies). We translate this type into a sized version, and also consider that the parameter `a` is a sized type:

```
data ListS : (Nat->#) -> Nat -> # where
   NilS : ListS A O
 | ConsS : {A:Nat -> #} -> (a:A n) -> (ListS A k) -> (ListS A (S k));
```

In this framework, functions return values as a `Size` structure. Consider the definition of the `append` function over lists, initially defined as follows:

```
append : List a -> List a -> List a;
append Nil ys = ys;
append (Cons x xs) ys = x (Cons xs ys);
```

Using the analyses we have developed for Hume, we can infer a size constraint for this function, namely that the size of the result is the sum of the sizes of the inputs. The function, when translated into the size framework, explicitly states this in its type, pairing the result with a proof of this size constraint (here, `\ _ . _` represents type-level abstraction):

```
appendS : {A:Nat->#} ->
          (ListS A xsn) -> (ListS A ysn) ->
          (Size (ListS A) (\n . \v . (n=(plus xsn ysn))));
appendS NilS ys = size ys ?;
appendS (ConsS x xs) = let (size val p) = append xs ys in
                           size (ConsS x val) ?;
```

Each of the clauses in this definition requires a property to be proved about the size constraints, indicated in the definition by `?`. The definition generates the following proof obligations:

```
?appendNil : ysn = plus O ysn;
?appendCons : (n = plus xsn ysn) -> ((S n) = plus (S xsn) ysn);
```

The first is solvable directly by evaluating (on type level) `plus O ysn`. To solve the second, we evaluate `plus (S xsn) ysn` and apply a proof that the successor function preserves equality (as in Section 2.2):

```
appendNil : ysn = plus O ysn;
appendNil = refl;

appendCons : (n = plus xsn ysn) -> ((S n) = plus (S xsn) ysn);
appendCons p = eq_resp_S p;
```

For size metrics associated with data structures, we have found the framework to be expressive. In particular, the framework allows higher order functions and here dependent types allow the cost metrics to be more expressive. A simple example is `twice`, which applies a function twice (i.e. it is the Church numeral 2):

```
twice f x = f (f x)
```

In this case, the cost of `twice` depends on the cost of `f` and how `f` is used. We therefore need additional parameters to represent this. There are three additional parameters: the cost of f (`fs`), the property it satisfies (`P`), and how that property is preserved by repeated application (`Ptrans`). In particular, note that the term `fs (fs as)` expresses that the costs of `f` are incurred twice.

```
twice : (P:(n:Nat)->(A n)->Nat->#) ->
        (fs:Nat -> Nat) ->
        (Ptrans:P as a (fs cs) -> P bs b (fs as) -> P bs b (fs (fs cs))) ->
        (f:A an' -> Size A (\n . \v . P n v (fs an'))) ->
        (A as) -> (Size A (\n . \v . P n v (fs (fs as))));
twice P fs Ptrans f x = let size val1 p1 = f x in
                        let size val2 p2 = f val1 in
                        size val2 ?;
```

Strengths and Weaknesses of this approach

We have implemented several examples in this framework, of varying complexity, including higher order functions such as `map` and `fold`. Other functions we have found less easy to express, such as `scan` (a fold which retains the intermediate values), where it is difficult to write the cost concisely because the intermediate values are retained in the result.

The framework itself is easy to implement. Our first implementation used the Coq theorem prover [Coq01], implementing specialised tactics to help with program construction and using the Omega decision procedure [Pug92] to solve any size constraints automatically. We have since reimplemented these ideas in our own language, Idris, as part of an investigation into implementing a dependently typed core language for Hume. The main strength of using dependent types is the ability to represent *higher order* costs, deriving cost formulae from the way in which higher order functions are applied.

While the framework itself is easy to implement, the translation from a source language into the framework is more difficult. So far, we have carried out this translation by hand, in order to understand the limitations of the framework. The translation of higher order functions is particularly challenging, although the types of the required predicate transformers were inferred automatically. The main weakness of the approach is the limitation in its expressivity: we have only been able to represent monotonically increasing costs of data structures, and the framework will require extensive changes in order to represent time or stack metrics.

3.2 Deep Embeddings

We have also investigated the *deep embedding* of costed programs in a dependent type theory. This involves the representation of the language and its type system as a dependently typed data structure, and an interpreter for this language. In this way, we would ultimately like to be able to represent the typing rules for our analyses *directly* in the implementation.

```
data Ty = TyNat | TyFun Ty Ty | TyBool;

data Term : (Vect Ty n) -> Ty -> # where
   Var : (i:Fin n) -> (Term G (vlookup i G))
 | Lam : (Term (VCons s G) t) -> (Term G (TyFun s t))
 | App : (Term G (TyFun s t)) -> (Term G s) -> (Term G t)
 | NatVal : Nat -> (Term G TyNat)
 | BoolVal : Bool -> (Term G TyBool)
 | Op : (op : (interpTy a) -> (interpTy b) -> (interpTy c)) ->
        (Term G a) -> (Term G b) -> (Term G c);
```

Figure 1: Simply typed λ-calculus representation

3.2.1 Language Representation

Figure 1 shows a data type which represents the abstract syntax of the simply typed λ-calculus, including variables (`Var`), λ-binding (`Lam`), function application (`App`), concrete values (`NatVal` and `BoolVal`) and primitive operators on those values (`Op`). The `Term` structure is parametrised over a type, `Ty`, and a context (as a vector of `Ty`), which allow the direct representation of the *typing rules* in the structure. In this way, we can only represent *well-typed* terms with the structure — we exploit Idris's type checker, rather than writing our own type checker for the object language. Additionally, it is possible to compute the concrete type of the interpretation of a `Term`.

3.2.2 Environments

To evaluate `Term`s, we will need to keep track of values stored in bound variables. The *environment* is a common pattern in dependently-typed programming. It implements a heterogeneous list (i.e. where each list element can be a different type), where the type is determined by a vector of representation types (c.f. `HList` in Haskell [KLS04]). In the code we define in this section, we have the implicit arguments:

```
using (iR:R -> #, xs:Vect R n)
```

We have a vector of representation types, `xs`, where `R` is a representation and `iR` is a function converting that representation into a concrete host language type. Now environments are defined as interpretations of a vector of representation types:

```
data Env : (R:#) -> (iR:R->#) -> (xs:Vect R n) -> # where
     Empty : (Env R iR VNil)
   | Extend : (res:(iR r)) -> (Env R iR xs) ->
              (Env R iR (VCons r xs));
```

For example, a simple language might have base types of natural numbers, booleans and the unit type. For these types, we can define an environment `SEnv` as follows:

```
data SimpleTy = SNat | SBool | SUnit;

interpSimple : SimpleTy -> #;
interpSimple SNat = Nat;
interpSimple SBool = Bool;
interpSimple SUnit = ();

SEnv = Env SimpleTy interpSimple;
```

An environment containing a `Nat` and a `Bool` can be written as:

```
sTy = VCons SNat (VCons SBool VNil);

sEnv : SEnv sTy;
sEnv = Extend (S (S O)) (Extend True Empty);
```

We will require functions for querying and manipulating environments. Since they are indexed over vectors, we will also need corresponding functions *in the type* to guarantee that the environment manipulation corresponds to the underlying manipulation of the representation vector. Looking up an item in an environment gives a value of the appropriate type:

```
envLookup : (i:Fin n) -> (Env R iR xs) -> (iR (vlookup i xs));
```

Given a function that updates a value in a vector, we can update a value in the environment. Since we are updating the representation vector, we do not need to preserve the type in the environment:

```
update : (i:Fin n) -> A -> (Vect A n) -> (Vect A n);
updateEnv : (Env R iR xs) -> (i:Fin n) -> (iR newR) -> (Env R iR (update i newR xs));
```

We will, however, need to extend environments when new resources are created:

```
snoc : (Vect A n) -> A -> (Vect A (S n));
addEnd : (Env R iR xs) -> (r:iR ty) -> (Env R iR (snoc xs ty));
```

Finally, when we are finished with a resource, it will be useful to remove it from the environment:

```
dropEnd : (Env R iR (snoc xs x)) -> (Env R iR xs);
```

The type of `dropEnd` explicitly states that the item to be dropped is the item at the end of the vector — it indicates clearly that it undoes the effect of `addEnd`.

3.2.3 Interpreter

Given an implementation of environments, we can now implement the interpreter for `Terms`. The definition is given in Figure 2. The interpreter directly translates terms in the object language to their Idris equivalent, having computed the relevant type with `interpTy`. Using dependent types, we avoid any need to tag the result with its type — we always know what type the interpretation of a program will be.

The language here is very simple, however. It is interesting to see that a type system can be written down directly, and the host language's type checker can be used to make sure that object language terms are well typed. We would like to see if we can take this idea further, and implement a *resource-aware* type system in this way, with its interpreter.

3.2.4 Cost Representation

We began to extend the well-typed interpreter with cost information of the form described in section 3.1. To represent a full language in this way proved difficult. We began by parameterising types by size variables, and representing this in the object language. We defined size expressions (with variables), as follows:

```
data Size : Nat -> # where
   SNum : Nat -> (Size n)
 | SVar : (i:Fin n) -> (Size n)
 | SOp : (Nat->Nat->Nat) -> (Size n) -> (Size n) -> (Size n);
```

```
interpTy : Ty -> #;
interpTy TyNat = Nat;
interpTy (TyFun s t) = (interpTy s)->(interpTy t);
interpTy TyBool = Bool;

interp : {G:Vect Ty n} -> (Env Ty # G) -> (Term G t) -> (interpTy t);
interp env (Var i) = envLookup i env;
interp env (Lam {s} sc) = \ v:(interpTy s) . (interp (Extend v env) sc);
interp env (App f a) = (interp env f) (interp env a);
interp env (NatVal n) = n;
interp env (BoolVal b) = b;
interp env (Op f l r) = f (interp env l) (interp env r);
```

Figure 2: Simply typed λ-calculus interpreter

Size expressions are paremeterised by the number of size variables abstracted over them. Types may be parameterised by sizes, for example in the simplest case, natural numbers could have a size representing their value (`TyNat`). We also include a binder (`TyBind`), for introducing size variables:

```
data Ty : Nat -> # where
   TyNat : (Size n) -> (Ty n)
 | TyFun : (Ty n) -> (Ty n) -> (Ty n)
 | TyBind : (Ty (S n)) -> (Ty n)
```

We can now, for example, include sized natural numbers in the langauge:

```
data Term : (G:TyEnv n) -> Ty -> # where
    ...
  | zero : Term G (TyNat (SNum O))
  | suc : (Term G (TyNat n)) -> (Term G (TyNat (S n)))
    ...
```

There are several advantages to taking this approach. Effectively, we are fully implementing the required type system in a theorem prover. Therefore we provide, together with the implementation, full, complete, sound proofs of the required properties. If we were able to do this for the whole of the Hume language, exploiting the analyses available to us, we would be able to build full machine checkable proofs of the extra-functional properties of Hume programs.

Unfortunately, we run into difficulties very quickly with this approach, which are difficult to overcome in the short term. For example:

- We need to add data structures (such as `Nat` above) explicitly to the `Term` structure, rather than in a generic way. Further research into generic programming with dependent types [MMA04] is required to make this practical.
- One of the benefits of the shallow embedding is that any host language expression can be used for costing. This is more difficult with the deep embedding, and one of the benefits of it (the representation of higher order functions) is therefore also lost.
- To get the full benefit of dependent types, it is helpful to be able to evaluate terms during type checking. This would require the definition of the interpreter simultaneously with the language, which is not yet supported by Idris.

While these are large problems which will require a significant amount of further research to overcome, we would still like to find a way to benefit from the well-typed interpreter approach, as it would allow us to reason about cost metrics other than heap usage. To this end, we have begun to explore other methods of implementing Domain Specific Embedded Languages (DSELs).

4 Resource Aware Domain Specific Languages

The most promising approach we have found for representing resource properties is based on the idea of *Domain Specific Embedded Languages* (DSELs). A domain-specific language (DSL) is a programming language dedicated to a specific problem domain. Using a DSL, a **domain expert**, who is not necessarily a computer programmer, can focus on details of the problem to be solved, rather than on details of the implementation language. A domain-specific *embedded* language [Hud96] is a DSL implemented by embedding the constructs into another language, the *host* language. In what follows, it is important to maintain a clear distinction between the implementation language (the **host language**) and the language being implemented (the **DSEL**). We also distinguish the DSEL implementer (developing a DSEL in the host language) from the DSEL programmer (a domain expert developing a program using the DSEL).

The main advantage of the DSEL approach is that it facilitates the development and deployment of a DSL by allowing the implementer to reuse host language features, such as the host language parser and code generator (and to some extent even the type system, as demonstrated by [Thi05, AMS08]). Because of its flexibility (especially in the type system), clarity and stability, Haskell has proved be to a popular choice of host language, and several DSELs have been previously implemented using it, e.g. [BCSS98, Thi05, AMS08]. Using a dependently typed language as the host language [OS08] allows us to take these ideas even further — we can specify the type system of the DSEL precisely in the host language, and impose any required constraints on constructs and values with dependent types.

An additional difficulty with both the shallow and deep embeddings we have described is the problem of representing *state*. The DSEL approach allows us to represent explicitly what happens during runtime, considering the state and using the type system to allow reasoning about state. Using this approach, we have implemented a simple memory management DSEL and considered how to apply it in a resource limited setting. We have also considered a stack management DSEL, and shown how it can preserve correctness properties of the stack and how stack size constraints can be embedded, and checked either statically or dynamically. Using the domain specific embedded language approach, we can exploit the expressive features of the host language where necessary, and concentrate on the required domain specific properties of the object language in the representation and the interpreter.

4.1 Resource framework

We now consider the structure of a domain-specific embedded language for managing resource states. We assume a resource type `R`, and define types in the language including an embedding of host language types, and an interpretation function:

```
data Ty = TyUnit | TyLift # | ... ;

interpTy : Ty -> #;
interpTy TyUnit = ();
interpTy (TyLift A) = A;
...
```

This type can be extended with any other required language types. Any host language type can be embedded as a `TyLift`. Figure 3 then gives a basis for defining *any* resource handling DSEL. This

```
using (ts:Vect R n, ts':Vect R n', tI:Vect R nI) {
  data Lang : (Vect R n)->(Vect R n')->Ty->* where
     ACTION : (action:IO ()) -> (Lang ts ts TyUnit)
   | RETURN : (val:A) -> (Lang ts ts (TyLift A))
   | BIND   : (code:Lang ts tI ty) ->
              (k:(interpTy ty) ->
              (Lang tI ts' tyout)) ->
              (Lang ts ts' tyout)
   ...
}
```

Figure 3: Skeleton resource-aware DSEL

assumes a resource type `R`, and defines generic operations for embedding host language values, returning DSEL values, and combining programs. `ACTION` executes an IO action, then returns the environment unchanged. `RETURN` injects a host language value into the DSEL, and `BIND` passes the result of one computation on to the next computation, updating the state in between — this threads the state through a computation in a manner similar to Atkey-style parameterised monads [Atk06].

4.2 Example — Global variables

The simplest example of a resource is a global variable. Let us consider a language with global variables defined in our framework, where the number of variables is a static constant. Variable types are defined with `SimpleTy` as above. We will replace `R` with `SimpleTy` and define `Ty` as follows:

```
data Ty = TyUnit | TyLift # | TyS SimpleTy ;
```

Finally, we add a typed *getter* and *setter* for global variables. Resources are referred to by their position in the vector, using an instance of `Fin`. It is only possible to `GET` a value within the bounds of the environment, and it is only possible to `SET` a value if there is a proof that we are setting it to the correct type.

```
data Lang : (Vect SimpleTy n) -> (Vect SimpleTy n') -> Ty -> * where
   ...
 | GET : (i:Fin n) ->
         (Lang ts ts (TyS (vlookup i ts)))
 | SET : (i:Fin n) -> (t:interpSimple T) ->
         (p:ElemIs i T ts) -> (Lang ts ts TyUnit)
```

To define an interpreter for this language, we require an environment of global variables (the resources), for which we will use `SEnv`, defined above. An interpreter takes an initial environment and a program, and returns a value and an updated environment. We define pairs for the interpreter result:

```
data I a b = MkPair a b;
```

The type of the interpreter shows that a language transforms an environment of representations of `ts` to an environment over `ts'`, returning a value of type `T`.

```
interp : (SEnv ts) -> (Lang ts ts' T) -> (IO (I (SEnv ts') (interpTy T)));
```

The generic operations are implemented as follows:

```
interp env (ACTION io)
     = do { io;
            return (MkPair env II); };
interp env (RETURN val) = return (MkPair env val);
interp env (BIND code k)
     = do { coderes <- interp env code;
            interpBind coderes k; };
```

Interpreting `ACTION` simply involves evaluating its argument then returning the environment unchanged. `RETURN` returns the environment unchanged, along with the given value. `BIND` is implemented with the following helper function, which passes the value returned by one computation to the next:

```
interpBind : (I (HEnv ts) A) -> (A -> (Lang ts ts' B)) -> (IO (I (HEnv ts') (interpTy B)));
interpBind (MkPair env val) k = interp env (k val);
```

The getter/setter are implemented with `envLookup`/`updateEnv` respectively. Note that although `SET` requires a proof as an argument, we do not need to inspect this proof — the very *existence* of the proof is enough to ensure that the value is the correct type to place in the environment.

```
interp env (GET i) = return (MkPair env (envLookup i env));
interp env (SET i v p) = return (MkPair (updateEnv env i v));
```

Using the environment `sEnv` defined above, we could write an example program which doubles the value of the first variable:

```
do using (BIND, RETURN) {
  double : Lang sTy sTy TyUnit;
  double = do { v <- GET f0;
                SET f0 (plus v v) ?;
              };
}
```

This uses a metavariable (`?`), generating a proof obligation. We need to show that the global `f0` is a `SNat`. The Idris environment accepts this program but will not execute it until the proof obligations are satisfied. The `:m` command (an abbreviation for "metavariables") allows us to identify the obligations that need to be proved :

```
Idris> :m
Proof obligations:
        [double_1]
```

We can now begin proving this lemma. The `:p` command starts the interactive proof mode:

```
Idris> :p double_1

--------------------------------
H0 ? (v : Nat) -> ElemIs f0 SNat
                    (VCons SNat (VCons SBool VNil))
```

This operates in a similar way to Coq, in that tactics can be used to construct a proof term which satisfies the goal. In this case, we need to prove an implication, so `intro` will introduce an assumption:

```
Idris> intro

v : Nat
--------------------------------
H0 ? ElemIs f0 SNat (VCons SNat (VCons SBool VNil))
```

The goal is now to show that the first element of the vector is `SNat`. Since it is, applying the `first` constructor of `ElemIs` will achieve this. We do this using `refine first`:

```
double_1> refine first

No more goals
```

When a proof is complete, the `qed` tactic verifies the proof term and outputs a trace, which can be pasted into the program.

```
double_1> qed
double_1 proof {
        %intro;
        %refine first;
        %qed;
};
```

5 A DSEL for Memory Management in Resource Limited Systems

The global variable example in the previous section outlines how we deal with state in our framework. This is, of course, no different from what can be achieved with a *state-transformer monad*. The real, additional, benefit of our approach comes from the ability to reason about state and to represent state constraints. If we are writing a low-level program, e.g. a device driver or embedded system controller, we would like to ensure the correct and safe use of memory, i.e.:

- All reads and writes must access memory locations which are allocated and in bounds.
- All memory must be freed before exit.

While adding dynamic checks for allocation success causes no additional difficulty, it will obscure our presentation. For the moment, we will therefore assume that all allocations succeed, i.e. that the underlying system has unlimited memory. We will consider this problem in the context of resource limited systems in Section 5.4.

5.1 Resources

We aim to implement a DSEL which manages system memory safely. To do this, we will require access to system calls. IDRIS has a foreign function interface which allows us to implement this, using `Ptr`, which is a foreign language pointer (a `void*` in C):

```
data Binary = mkBin Ptr;

malloc  : Int    -> (IO Binary);
free    : Binary -> (IO ());
getByte : Binary -> Int -> (IO Int);
putByte : Binary -> Int -> Int -> (IO ());
```

The DSEL programmer has no direct access to these primitive actions (which have the obvious effects), rather, they must be performed through the operations of the DSEL. As with the global variable DSEL above, we begin by defining resource types. Resources are blocks of memory, which are either available or free. If they are available, they have a size:

```
data MemState = Available Int | Freed;
```

If a block is available, it will also be associated with a pointer. The interpretation of a resource type should include this pointer:

```
data MemHandle : MemState -> # where
   Alloc : (h:Binary) -> (MemHandle (Available size))
 | Dealloc : MemHandle Closed;
```

Together `MemState` and `MemHandle` are the *representation type* and *interpretation* of resource states respectively. This allows us to define environments of handles as follows:

```
HEnv = Env MemState MemHandle;
```

An environment is indexed by a vector of resource states, which we will refer to as the *resource vector*. When we use a memory reference, we will need to know that it has been allocated and is available. An instance of the predicate `IsAlloc` proves this:

```
data IsAlloc : (Fin n) -> (Vect MemState n) -> # where
   allocFirst : (IsAlloc fO (VCons (Available size) xs))
 | allocLater : (p:IsAlloc i xs) ->
                (IsAlloc (fS i) (VCons x xs));
```

Memory accesses must be within the bounds of allocated blocks:

```
data InBound : Int -> (Fin n) ->
               (Vect MemState n) -> # where
   boundFirst : (so (s<p)) ->
                (InBound s fO (VCons (Available p) xs))
 | boundLater : (p:InBound s i xs) ->
                (InBound s (fS i) (VCons x xs));
```

A proof of `InBound` makes use of the following predicate:

```
data so : Bool -> # where oh : so True;
```

The primary use of `so` is to demonstrate that a dynamic check has been performed, resulting in a boolean value. Using `so`, we not only know statically that the check (or a related check) must have been run, but also that we *need not check again*. In this case, it shows that there has been a check that the address we would like to use is within the bounds of the available memory region. If a reference is within bounds (as proved by `InBound`), then it must have been successfully allocated. We can prove this by implementing the following function:

```
isAlloc : (InBound s i xs) -> (IsAlloc i xs);
```

5.2 Definition of the Memory Management DSEL

Operations in our DSEL will either return a memory handle, the unit value, or a value in the host language:

```
data Ty = TyHandle Nat | TyUnit | TyLift #;

interpTy : Ty -> #;
interpTy (TyHandle n) = Fin n;
interpTy TyUnit = ();
interpTy (TyLift A) = A;
```

```
data Lang : (Vect MemState n)->(Vect MemState n')->Ty->* where

{- Generic operations -}

   ACTION : (action:IO ()) -> (Lang ts ts TyUnit)
 | RETURN : (val:A) -> (Lang ts ts (TyLift A))
 | FORGET : (Lang ts (snoc ts' Freed) t) -> (Lang ts ts' t)
 | CALL   : (Lang VNil VNil t) -> (Lang ts ts t)
 | BIND   : (code:Lang ts tI ty)-> (k:(interpTy ty)->(Lang tI ts' tyout)) -> (Lang ts ts' tyout);

{- Domain-specific operations -}

 | ALLOC  : (size:Int) -> (Lang ts (snoc ts (Available size)) (TyHandle (S n)))
 | FREE   : (i : Fin n) -> (IsAlloc i ts) -> (Lang ts (update i Freed ts) TyUnit)
 | PEEK   : (i:Fin n) -> (loc:Int) -> (b:InBound loc i ts) -> (Lang ts ts (TyLift Int))
 | POKE   : (i:Fin n) -> (loc:Int) -> (val:Int) -> (b:InBound loc i ts) -> (Lang ts ts TyUnit)
```

Figure 4: Memory-management DSEL

Any handle created by the DSEL must record the number of handles that were available at the point of creation. Having defined resources and their constraints, and the types of DSEL programs, we are now in a position to write the language specification in a way which satisfies the requirements on memory safety that were given at the start of this section. The DSEL is indexed over the input and output resources, and its return type. It is convenient to name all implicit arguments for the resources in a `using`-clause:

```
using (ts:Vect MemState n,
       ts':Vect MemState n',
       tI:Vect MemState nI) {
  data Lang : (Vect MemState n) ->
              (Vect MemState n') -> Ty -> * where
  ...
}
```

The full definition of the DSEL is given in Figure 4. Note, in particular, the operations which use memory: `FREE` ensures that the given handle is allocated, and `PEEK` and `POKE` require proofs of `InBound` in order to guarantee that the location given is within the bounds of an allocated region. A *memory-safe* program is then a program which satisfies all of the required constraints on memory usage, and which has freed all memory handles on termination. We can write this directly as:

```
MemSafe : Ty -> *;
MemSafe T = Lang VNil VNil T;
```

Note that on entry to the program, and when it exits, no resources are in use. This is expressed by the `VNil` resource vectors. Now that we have a language in which to manage memory, the soundness of memory management rests entirely on the correct representation of the appropriate constraints in `Lang`. *No further type soundness proofs are required.*

Creating and destroying resources

Unlike the simple global variable DSEL, programs in the memory-management DSEL can create and destroy resources. We also allow programs to call other programs in the DSEL, an essential requirement

```
interp : (HEnv ts) -> (Lang ts ts' T) -> (IO (I (HEnv ts') (interpTy T)));

interpBind : (I (HEnv ts) A) -> (A -> (Lang ts ts' B)) -> (IO (I (HEnv ts') (interpTy B)));
interpBind (MkPair env val) k = interp env (k val);

{- Generic operations -}

interp env (ACTION io) =        do { io;
                                     return (MkPair env II); };
interp env (RETURN val) =       do { return (MkPair env val); };
interp env (FORGET p) =         do { runp <- interp env p;
                                     return (dropLast runp); };
interp env (CALL p) =           do { runp <- interp Empty p;
                                     return (anyEnv env runp); };
interp env (BIND code k) =      do { coderes <- interp env code;
                                     interpBind coderes k; };

{- Domain specific operations -}

interp env (ALLOC size) =       do { fh <- malloc size;
                                     return (MkPair (addEnd env (Alloc fh)) bound); };
interp env (FREE i p) =         do { free (getMem p env);
                                     return (MkPair (updateEnv env i Dealloc) II); };
interp env (PEEK i loc p) =     do { val <- getByte (getMem (isAlloc p) env) loc;
                                     return (MkPair env val); };
interp env (POKE i loc val p) = do { putByte (getMem (isAlloc p) env) loc val;
                                     return (MkPair env II); };
```

Figure 5: Memory-management DSEL interpreter

for larger-scale systems. Let us look at how this is exploited in practice. The memory allocation command has the following type:

```
ALLOC : (size:Int) ->
        (Lang ts (snoc ts (Available size))
             (TyHandle (S n)))
```

This command adds a new, available, resource to the list, and returns a handle that is parameterised by the number of resources that were available at creation time. This is necessary because the total number of resources is bounded. We represent this with a `Fin` type. Although the type does not express it, the resource returned will always be the largest value in `Fin (S n)`, representing the state at the end of the resource vector. We add resources to the end of the list, rather than the start, so that any resource identifiers that have already been created will never be invalidated.

As well as allocating resources, we can also destroy them. We use the generic operation `FORGET` to remove a resource from the *end* of the list when it has been freed. Note that we cannot remove resources from the middle of the list, as this would invalidate resource handles. The main use of `FORGET` is to tidy up a program's type, so that it can be invoked by `CALL`.

5.3 Interpreter Definition

The full interpreter for `Lang` is given in Figure 5. This is, in general, a straightforward extension of the interpreter for the global variable language. The interpreter requires some helper functions. Since their

definitions are straightforward, we give only their types here. Given a proof that a resource handle represents allocated memory in an environment, we can retrieve the pointer to that memory:

```
getMem : (IsAlloc i xs) -> (HEnv xs) -> Binary;
```

We can always remove the last value from an environment, which allows us to implement `FORGET`:

```
dropLast : (I (HEnv (snoc ts x)) (interpTy t)) ->
           (I (HEnv ts) (interpTy t));
```

We can always pair a value with an arbitrary environment, which allows us to implement `CALL`:

```
anyEnv : (HEnv ts) -> (I (HEnv VNil) (interpTy t)) ->
         (I (HEnv ts) (interpTy t));
```

5.4 Allocation certificates

In the context of resource-limited systems, we can use the resource framework to keep track of the total amount of allocation and to check that it stays below the limit of available memory.

```
data Lang : (Vect MemState n) -> Int -> -- before
            (Vect MemState n') -> Int -> -- after
            Int -> -- total memory available
            Ty -> #;
```

The `ALLOC` operation now requires a static check that it is not attempting to allocate beyond the end of the available memory:

```
ALLOC  : (size:Int) ->
         (p:so (mem_in+size<total_memory)) ->
         (Lang ts mem_in
            (snoc ts (Available size)) (mem_in+size)
              total_memory
             (TyHandle (S n)));
```

The argument `p` is a *certificate* (a proof term) that the total allocation remains within the limit available on the system. This may be calculated dynamically — in which case we may even choose a simpler form of certificate in which we merely perform a system call to check that the memory is available — or proved statically. A memory-safe program is now memory-safe with respect to a limit:

```
MemSafe : Int -> Ty -> #;
MemSafe total T = Lang VNil 0 VNil x total T;
```

5.5 Stack correctness

The power of the dependently typed DSEL approach to cost representation is that we can give a precise description of how the program *state* changes throughout execution, and as a result reason about the constraints which the state must satisfy. In the case of the memory management DSEL described above, we can decide whether an allocation is safe based on knowledge (static or dynamic) of exactly how much heap space is available.

We have also considered applying this method to verifying stack costs. In order to verify the stack-correctness of a program, we will need to consider the execution model. For the initial implementation of the DSEL, we consider a language running on a simple stack machine, where a computation (e.g. a function call) results in a value being pushed onto the stack, possibly with intermediate stack manipulations. Such a language might include commands such as the following:

```
data Lang : (Vect Ty n) -> (Vect Ty n') -> # where
 ...
 | PUSH : (val:interpTy T) -> (Lang ts (VCons T ts))
 | POP: (Lang (VCons T ts) ts)
 | ADD: (Lang (VCons TyNat (VCons TyNat ts))) -> (Lang (VCons TyNat ts))
 ...
```

`PUSH` and `POP` add and remove values to and from the stack respectively, and the type in each case reflects this. `ADD` requires there to be two numbers at the top of the stack, and results in one number. A stack-correct program then has the type:

```
StackSafe : Ty -> #;
StackSafe T = Lang VNil (VCons T VNil);
```

In a memory limited system, an additional constraint we would like to guarantee is that the stack size never exceeds a certain bound. We would therefore like to extend our representation of state to express this. As with the memory DSEL, we give the maximum allowed stack size as an additional parameter (this time as a `Nat` as vectors are indexed by `Nats`). Whenever the stack size changes, we need to check that it remains within bounds. A convenient place to check this is the `BIND` operation, since it is here that all state transformations are combined.

```
data Lang : (Vect Ty n) -> (Vect Ty n') -> Nat -> # where
 ...
 | BIND   : (code:Lang ts tI max)-> (k:(interpTy ty)->(Lang tI ts' max)) ->
            (p:LE n max) -> (Lang ts ts' max);
```

Again, these proofs can be computed statically or dynamically. Where possible, a static proof would mean that *no run-time check* would be necessary to guarantee that a system does not overflow the stack. If it is not possible, requiring the proof term to be computed dynamically would at least ensure that stack overflow errors were handled safely. Note that there is no overhead associated with the dynamic check other than the check itself, as the run-time representation of the proof-term is erasable, as described by [BMM03, Bra05].

6 Summary

We have implemented a prototype dependently typed computation language, Idris, and used it for experimenting with the representation of costs of functional programs, such as the expression layer of Hume. We have investigated several methods for cost representation to establish the feasibility of using dependent types as a representation of a full-scale resource aware programming language.

We have found that there is a tradeoff between expressivity of the representation and ease of implementation and translation from a source notation. This is not surprising — a more expressive representation capturing more resource properties and more constraints will require more justification, in the form of a machine checkable proof term, that those constraints are satisfied. With the shallow embedding of costs in the host language described in Section 3.1, it is relatively straightforward to construct costed programs, but is limited to heap costs. Nevertheless, we have found this method allows a detailed representation of heap costs, including the costs of higher order functions. Programs in this setting are fully and independently machine checkable.

A full representation as a deep embedding, as described in Section 3.2, would overcome the limitations of the shallow embedding but requires a lot of work to implement well. With a deep embedding, we get full control over the form of the language, its representation and its evaluation. To achieve the

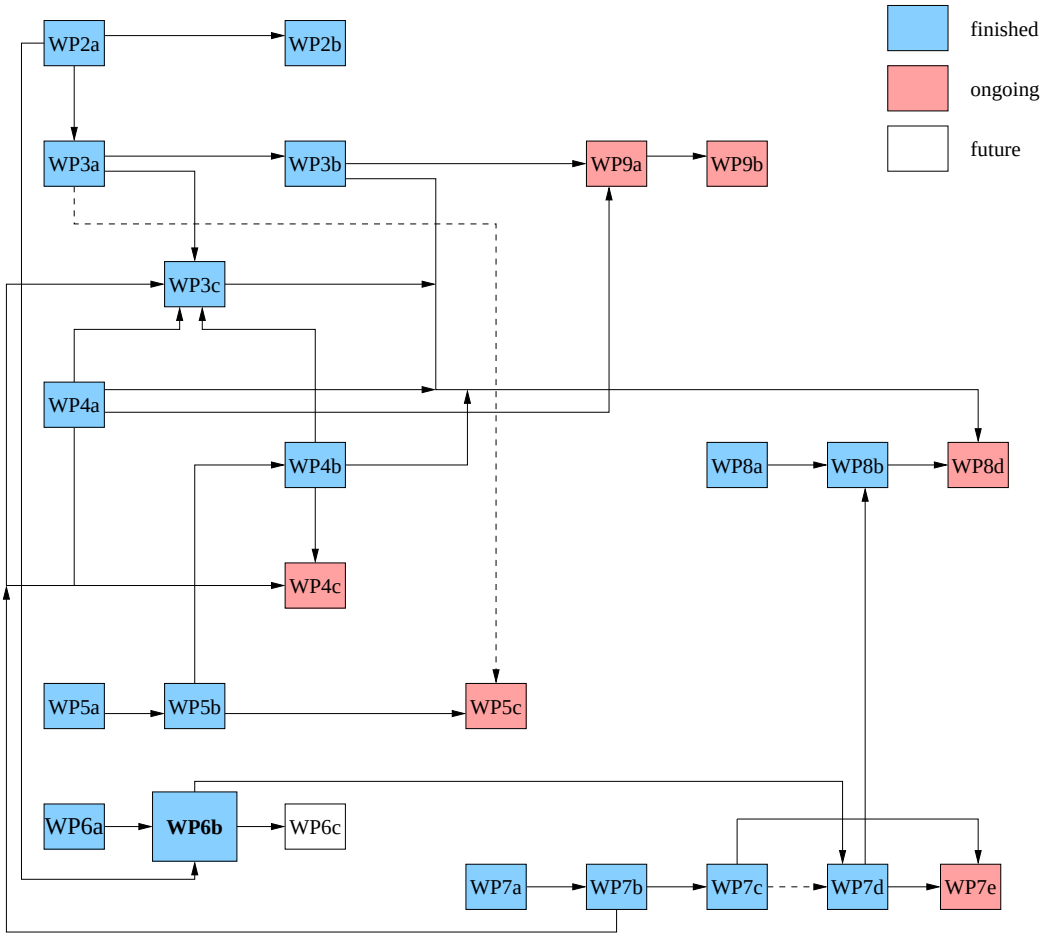

Figure 6: Project Task Dependencies

representation of all of our analyses and the resource logic within this, however, will require further development of the host language.

The most effective approach we have found to be a domain specific embedded language approach, which attempts to combine features of the shallow and deep embeddings. By embedding host language terms in the DSEL, we can construct any required resource constraint, and we can use host language features to help with variable binding and evaluation. By implementing this as separate abstract syntax, with an interpreter, we can add any features we need which are not part of the host language, such as stack manipulation, and embed proofs of the necessary constraints. We envisage using this method to implement an intermediate language for Hume programs, capturing the properties of the target machine in a dependent type, and inserting proof terms where possible and dynamic checks where necessary.

Further work is required to map real Hume programs into this setting, and there are several research questions to answer. For example, how much can be proved statically, and how much will we need to rely on dynamic checking? How can we represent all of the necessary resource constraints? Could we potentially use the Idris DSEL as an execution environment? Our initial investigation into the implementation of the resource aware DSEL approach suggests that it is a promising line of research into proving the extra-functional correctness of resource limited systems.

6.1 Positioning of this Deliverable

Figure 6 gives an overview of the dependencies between workpackages, with the current deliverable D42 in WP6d highlighted. This deliverable makes use of the analyses for resource consumption, described in Deliverables D05 (WP4a) [JLH07b], D11 (WP4b) [JLH07a] and D14 (WP3a) [JLH07c]. These deliverables describe how to automatically infer information on heap-space consumption, stack-space consumption and worst-case execution time, whereas this deliverable focuses on verifying such bounds in a dependently typed language. This work is related to Deliverable D21 (WP6b) [LB08], which defines a format of certificates for bounded resource consumption in the form of proofs in a specialised resource logic. As in this deliverable, the trace of a proof or a proof term can be used as a certificate. The main difference is, that in D21 we formalise a resource logic in a general theorem prover, Isabelle/HOL, and profit from the rich expressive power of the underlying metalanguage. In this deliverable we use a dependently typed language to formalise resource constraints. This language provides a higher level of abstraction and is therefore better suited for encoding resource constraints, however, its expressivity is limited. The studies of different forms of embeddings try to find a balance between these requirements. Most notably, the formalisation of resource bounds through dependent types in this deliverable gives us soundness of the bounds for free, i.e. it follows from the general soundness result of the dependent type system. In contrast, our formalisation of the resource logic in D21 required (a fairly complicated) soundness proof in Isabelle/HOL.

In summary, we observe that the type-based approach promoted in this deliverable offers the attractive prospect of implicitly verified bounds, expressed as dependent types in an intermediate language, but further research is needed to formalise general resource bounds of the Hume expression layer. For the time being, the Isabelle/HOL encoding of a resource logic for the Hume expression layer is the more direct approach to certify bounded resource consumption, based on the results from the existing, automatic resource inference.

References

[AMS08] Lennart Augustsson, Howard Mansell, and Ganesh Sittampalam. Paradise: a Two-stage DSL Embedded in Haskell. *SIGPLAN Not.*, 43(9):225–228, 2008. 4

[Atk06] Robert Atkey. Parameterized Notions of Computation. In *MSFP 2006: Proceedings of the Workshop on Mathematically Structured Functional Programming*, 2006. 4.1

[BCSS98] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware Design in Haskell. In *International Conf. on Functional Programming (ICFP 1998)*, pages 174–184, Baltimore, Maryland, United States, 1998. ACM. 4

[BH06] Edwin Brady and Kevin Hammond. A Dependently Typed Framework for Static Analysis of Program Execution Costs. In *International Workshop on Implementation and Application of Functional Languages (IFL 2005)*, volume 4015 of *LNCS*, pages 74–90, Dublin, Ireland, September 19–21, 2006. Springer. 3.1

[BMM03] Edwin Brady, Conor McBride, and James McKinna. Inductive families need not store their indices. In *Types for Proofs and Programs (TYEPS 2003)*, volume 3085 of *LNCS*, pages 115–129, Torino, Italy, April 30 – May 4, 2003. Springer. 5.5

[Bra05] Edwin Brady. *Practical Implementation of a Dependently Typed Functional Programming Language*. PhD thesis, University of Durham, 2005. 5.5

[Bra07] Edwin Brady. Ivor, a Proof Engine. In *International Workshop on Implementation and Application of Functional Languages 2006 (IFL'06)*, volume 4449 of *LNCS*, pages 145–162, Budapest, Hungary, September 4-6, 2007. Springer. 2

[Coq01] Coq Development Team. The Coq proof assistant — reference manual. `http://coq.inria.fr/`, 2001. 3.1

[Hud96] Paul Hudak. Building Domain-specific Embedded Languages. *ACM Computing Surveys*, 28A(4), December 1996. 4

[JLH07a] S. Jost, H-W. Loidl, and K. Hammond. Report on Heap-space Analysis. EmBounded Project Deliverable, February 2007. Deliverable D11. 1, 1.1, 6.1

[JLH07b] S. Jost, H-W. Loidl, and K. Hammond. Report on Stack-space Analysis. EmBounded Project Deliverable, February 2007. Deliverable D5. 1, 1.1, 6.1

[JLH07c] S. Jost, H-W. Loidl, and K. Hammond. Report on WCET Analysis. EmBounded Project Deliverable, February 2007. Deliverable D14. 1, 1.1, 6.1

[KLS04] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly Typed Heterogeneous Collections. In *Haskell '04: Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 96–107, Snowbird, Utah, USA, 2004. ACM Press. 3.2.2

[LB08] H-W. Loidl and L. Beringer. Certificates. EmBounded Project Deliverable, July 2008. Deliverable D21. 1, 1.1, 6.1

[MMA04] Peter Morris, Conor McBride, and Thorsten Altenkirch. Exploring the Regular Tree Types. In *Types for Proofs and Programs (TYPES 2004)*, volume 3839 of *LNCS*, pages 252–267, Jouy-en-Josas, France, December 15–18, 2004. 3.2.4

[OS08] Nicolas Oury and Wouter Swierstra. The Power of Pi. In *International Conf. on Functional Programming (ICFP 2008)*, pages 39–50, Victoria, BC, Canada, September 20–28, 2008. ACM. 4

[Pug92] William Pugh. The Omega Test: a Fast and Practical Integer Programming Algorithm for Dependence Analysis. *Communication of the ACM*, pages 102–114, 1992. 3.1

[Thi05] Peter Thiemann. An Embedded Domain-specific Language for Type-safe Server-side Web Scripting. *ACM Trans. Internet Technol.*, 5(1):1–46, 2005. 4